Capítulo QUATORZE

Classe Optional

Objetivos do Exame:
Desenvolver código que use a classe Optional.

O problema com null

A maioria das linguagens de programação tem um tipo de dado para representar a ausência de um valor, e ele é conhecido por muitos nomes:

NULL, **nil**, **None**, **Nothing**

O tipo *null* foi introduzido em ALGOL W por Tony Hoare em 1965, e é considerado um dos piores erros da ciência da computação. Nas próprias palavras de Tony Hoare:

Eu o chamo de meu erro de um bilhão de dólares. Foi a invenção da referência nula em 1965. Naquele tempo, eu estava projetando o primeiro sistema de tipos abrangente para referências em uma linguagem orientada a objetos (ALGOL W). Meu objetivo era garantir que todo uso de referências fosse absolutamente seguro, com verificação realizada automaticamente pelo compilador. Mas eu não resisti à tentação de incluir uma referência nula, simplesmente porque era muito fácil de implementar. Isso levou a inúmeros erros, vulnerabilidades e falhas de sistema, que provavelmente causaram um bilhão de dólares em dor e prejuízo nos últimos quarenta anos.
– Tony Hoare

Ainda assim, alguns podem estar se perguntando, qual é o problema com *null*?

Bem, se você está um pouco preocupado com os problemas que esse código pode causar, você já sabe a resposta:

```java
String summary =
  book.getChapter(10)
          .getSummary().toUpperCase();
```

O problema com esse código é que, se qualquer um desses métodos retornar uma referência *null* (por exemplo, se o livro não tiver um décimo capítulo), uma **NullPointerException** (a exceção mais comum em Java) será lançada em tempo de execução, interrompendo o programa.

O que podemos fazer para evitar essa exceção?

Talvez, a maneira mais fácil seja verificar *null*. Aqui está uma forma de fazer isso:

```java
String summary = "";
if(book != null) {
    Chapter chapter = book.getChapter(10);
    if(chapter != null) {
        if(chapter.getSummary() != null) {
          summary = chapter.getSummary()
                        .toUpperCase();
        }
    }
}
```

Você não sabe se algum objeto nessa hierarquia pode ser *null*, então você verifica cada objeto por esse valor. Obviamente, essa não é a melhor solução; ela não é muito prática e prejudica a legibilidade.

Pode haver outro problema. Verificar *null* é realmente desejável? Quero dizer, e se esses objetos nunca deveriam ser *null*? Ao verificar *null*, estaremos escondendo o erro e não lidando com ele.

Claro, isso também é uma questão de design. Por exemplo, se um capítulo ainda não tem um resumo, o que seria melhor usar como valor padrão? Uma string vazia ou *null*?

Para lidar com esse problema, o Java 8 introduziu a classe **java.util.Optional<T>**.

A classe Optional

A função desta classe é **ENCAPSULAR** um valor opcional, um objeto que pode ser *null*.

Usando o exemplo anterior, se sabemos que nem todos os capítulos têm um resumo, em vez de modelar a classe assim:

```java
class Chapter {
    private String summary;
    // Outros atributos e métodos
}
```

Podemos usar a classe Optional:

```java
class Chapter {
    private Optional<String> summary;
    // Outros atributos e métodos
}
```

Assim, se houver um valor, a classe **Optional** apenas o encapsula. Caso contrário, um valor vazio é representado pelo método Optional.empty(), que retorna uma instância singleton de Optional.

Usando esta classe em vez de *null*, primeiro, declaramos explicitamente que o atributo *summary* é opcional. Depois, podemos evitar **NullPointerExceptions** e ainda contar com os métodos úteis de Optional, que veremos a seguir.

Criando instâncias de Optional

Para obter um objeto Optional vazio, use:

```java
Optional<String> summary = Optional.empty();
```

Se você tem certeza de que um objeto **não** é *null*, pode encapsulá-lo em um Optional assim:

```java
Optional<String> summary = Optional.of("Um resumo");
```

Uma **NullPointerException** será lançada se o objeto for *null*. No entanto, você pode usar:

```java
Optional<String> summary = Optional.ofNullable("Um resumo");
```

Isso retorna uma instância Optional com o valor especificado, se ele não for *null*. Caso contrário, retorna um Optional vazio.

Verificando se Optional contém valor

```java
if( summary.isPresent() ) {
    // Faça algo
}
```

Ou, de forma mais funcional:

```java
summary.ifPresent(s -> System.out.println(s));
// Ou summary.ifPresent(System.out::println);
```

O método ifPresent() recebe um Consumer<T> como argumento, que é executado apenas se o Optional contiver um valor.

Obtendo o valor de um Optional

```java
String s = summary.get();
```

Porém, esse método lançará uma **java.util.NoSuchElementException** se o Optional não contiver valor. Portanto, é melhor usar o método ifPresent().

Métodos alternativos para valores padrão ou exceção

```java
String summaryOrDefault = summary.orElse("Resumo padrão");
```

orElse() retorna o argumento (que deve ser do tipo T, neste caso uma String) se o Optional estiver vazio. Caso contrário, retorna o valor encapsulado.

```java
String summaryOrDefault = summary.orElseGet(() -> "Resumo padrão");
```

orElseGet() recebe um Supplier<? extends T> que retorna um valor quando o Optional está vazio. Caso contrário, retorna o valor encapsulado.

```java
String summaryOrException = summary.orElseThrow(() -> new Exception());
```

orElseThrow() recebe um Supplier<? extends X>, onde X é o tipo da exceção a ser lançada quando o Optional estiver vazio. Caso contrário, retorna o valor encapsulado.

Optional para primitivos

Como *streams*, há versões de Optional para tipos primitivos: OptionalInt, OptionalLong e OptionalDouble, então você pode usar OptionalInt em vez de Optional<Integer>:

```java
OptionalInt optionalInt = OptionalInt.of(1);
int i = optionalInt.getAsInt();
```

No entanto, o uso dessas versões primitivas não é encorajado, principalmente porque elas não têm três métodos úteis da classe Optional: filter(), map() e flatMap(). E como Optional contém apenas um valor, o custo de *boxing/unboxing* de um primitivo não é significativo.

---

filter()

Retorna o Optional se um valor estiver presente e corresponder ao predicado fornecido. Caso contrário, retorna um Optional vazio:

```java
String summaryStr = summary.filter(s -> s.length() > 10)
                           .orElse("Resumo curto");
```

---

map()

Geralmente usado para transformar de um tipo para outro:

```java
int summaryLength = summary.map(s -> s.length())
                           .orElse(0);
```

---

flatMap()

Semelhante ao map(), mas recebe um Function<? super T, Optional<U>>. Se o valor estiver presente, retorna o Optional resultante da função. Caso contrário, retorna um Optional vazio.

No Capítulo 17, revisaremos com mais detalhes os métodos map() e flatMap() e como são usados com *streams*.

Pontos-chave

- A classe **java.util.Optional<T>** encapsula um valor opcional, ou seja, um objeto que pode ser *null*.
- Um valor vazio é representado pelo método Optional.empty().
- Você pode encapsular um objeto com Optional.of(), mas ele lançará uma **NullPointerException** se o objeto for *null*.
- O método ofNullable() retorna um Optional com o valor especificado se ele não for *null*, caso contrário, retorna vazio.
- get() retorna o valor, mas pode lançar **NoSuchElementException**. É preferível usar ifPresent().
- orElse(), orElseGet() e orElseThrow() são alternativas para fornecer um valor ou lançar uma exceção quando vazio.

Autoavaliação

1. Dado:

```java
public class Question_14_1 {
    public static void main(String[] args) {
        Optional opt = Optional.of("1");
        String s = opt.orElseGet(
            () -> new RuntimeException()
        );
        System.out.println(s);
    }
}
```

Qual é o resultado?

A. 1
B. Nada é impresso
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução

---

2. Qual das seguintes afirmações é verdadeira?

A. O método Optional.isPresent() recebe um Consumer<T> como argumento que é executado apenas se o Optional contiver um valor.
B. O método Optional.of() pode criar um Optional vazio.
C. O método Optional.of() pode lançar uma NullPointerException.
D. O método Optional.ifPresent() recebe uma Function<T,U> como argumento.

---

3. Dado:

```java
public class Question_14_3 {
    public static void main(String[] args) {
        System.out.println(ToInt("a").get());
    }
    private static Optional<Integer> ToInt(String s) {
        try {
            return Optional.of(Integer.parseInt(s));
        } catch(Exception e) {
            return Optional.empty();
        }
    }
}
```

Qual é o resultado?

A. a
B. Optional.empty
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução

---

4. Dado:

```java
public class Question_14_4 {
    public static void main(String[] args) {
        System.out.println(
            Optional.of(0).orElse(1)
        );
    }
}
```

Qual é o resultado?

A. 0
B. 1
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução